CSE 504: Compiler Design

Fall 2026

R. Sekar

1. Text Processing is Important

- Unstructured textual data is everywhere
  - Web pages and documents
  - Logs and diagnostic output
  - Natural-language text and messages
- Yet, fewer and fewer CS programs cover this topic
  - Using regular expressions to describe lexical structure
  - Using grammars to describe language syntax
  - Extracting information from logs and reports
  - Filtering and transforming text with sed and AWK
- Applications far beyond traditional compilers

2. Theory to Practice

Compilers harness the beauty of theoretical computer science to solve very concrete software engineering problems.

- *Automatic generation* of efficient implementations from compact, high-level specifications based on:
  - Regular and context-free languages
  - Automata and parsing algorithms
  - Type systems and program analysis
  - Machine architecture and code generation
- 50 year old research, still every bit as applicable as then!

3. Managing Complex Software Design and Implementation

- Modern compilers: among the most complex software systems
- Techniques for managing complexity
  - Modularize the design
  - Make modules flexible and extensible
  - Decouple design and implementation
  - Build efficient yet readable and maintainable implementations
- Experience with modern, object-oriented C++
  - The language of choice for much high-performance software

Translation Strategy

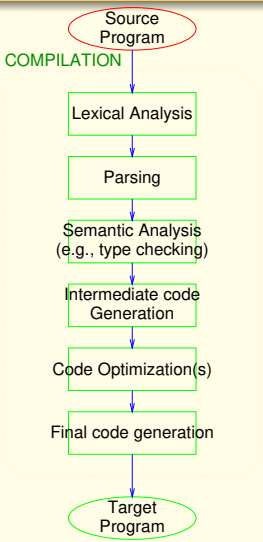
Classic Software Engineering Problem

- **Objective:** Translate a program in a high level language into *efficient* executable code.
- **Strategy:** Divide translation process into a series of phases.

  Each phase manages some particular aspect of translation.

Interfaces between phases governed by specific intermediate forms.

Translation Steps



Syntax Analysis Phase: Recognizes "sentences" in the program using the *syntax* of the language

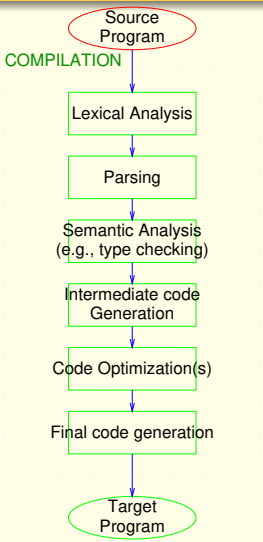
Semantic Analysis Phase: Infers information about the program using the *semantics* of the language

Intermediate Code Generation Phase: Generates "abstract" code based on the syntactic structure of the program and the semantic information from Phase 2.

Optimization Phase: Refines the generated code using a series of *optimizing* transformations.

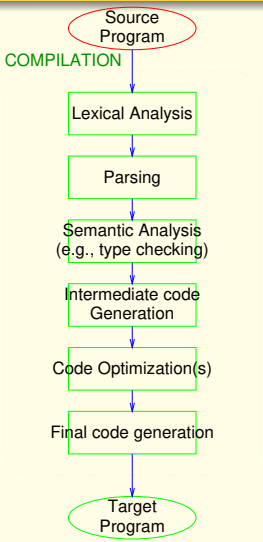
Final Code Generation Phase: Translates the abstract intermediate code into specific machine instructions.

Translation Steps: Lexical Analysis (Scanning Phase)



- Convert the *stream of characters representing input program* into a sequence of *tokens*.
- Tokens are the "words" of the programming language.
- For instance, the sequence of characters "`static int`" is recognized as two tokens, representing the two words "static" and "int".
- The sequence of characters "`*x++`" is recognized as three tokens, representing "*", "x" and "++".

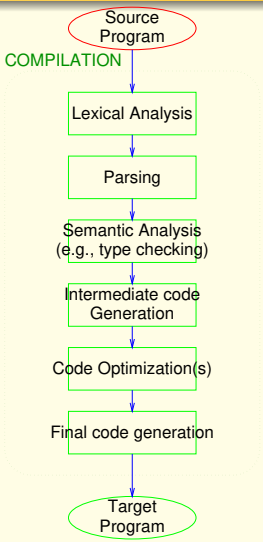
Translation Steps: Parsing (Syntax Analysis Phase)



- Uncover the *structure* of a sentence in the program from a stream of *tokens*.
- For instance, the phrase "x = -y", which is recognized as four tokens, representing "x", "=" and "-" and "y", has the structure **=(x, -(y))**, i.e., an assignment expression, that operates on "x" and the expression "-(y)".
- Build a *tree* called a *parse tree* that reflects the structure of the input sentence.

Typically, compilers build an *abstract syntax tree* directly, skipping the construction of parse trees.

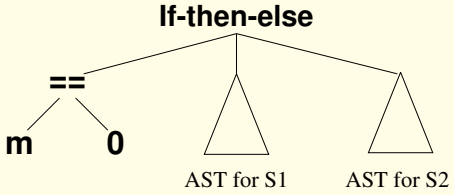
Translation Steps: Abstract Syntax Tree (AST)



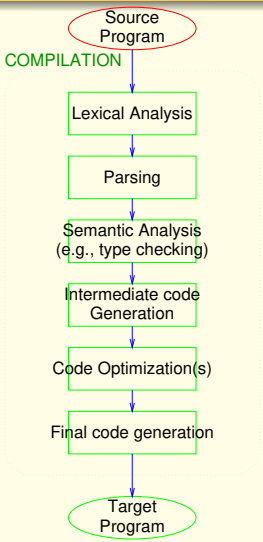
- Represents the syntactic structure of the program, hiding a few details that are irrelevant to later phases of compilation.
- For instance, consider a statement of the form:

```
if (m == 0) S1 else S2
```

where `S1` and `S2` stand for some block of statements. A possible AST for this statement is:

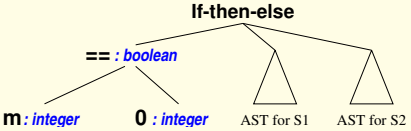


Translation Steps: Type Checking (Semantic Analysis)

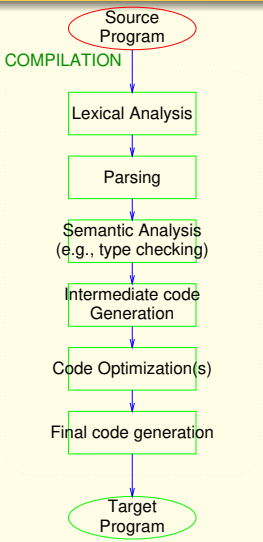


- Decorate the AST with semantic information that is necessary in later phases of translation.
- For instance, the AST

becomes

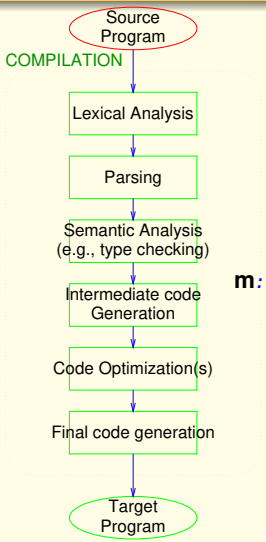


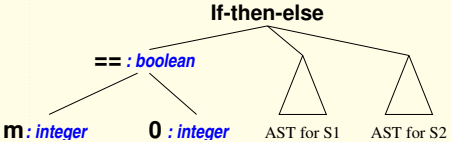
Translation Steps: Intermediate Code Generation



- Translate each sub-tree of the decorated AST into *intermediate code*.
- Intermediate code hides many machine-level details, but has instruction-level mapping to many assembly languages.
- Main motivation: a single representation that forms the basis of common optimizations across different front-end languages and backend architectures.

Translation Steps: Intermediate Code Generation Example

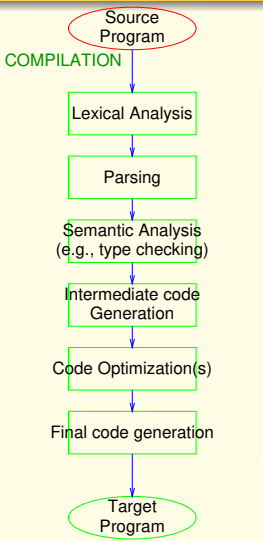




becomes

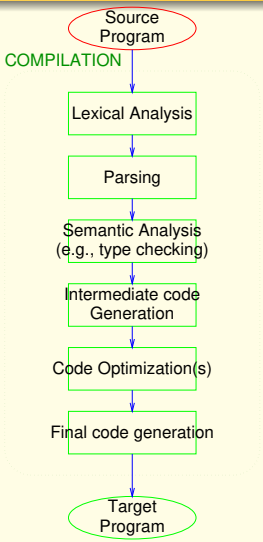
```
        R1 ← mem(m)
        cmp R1, 0
        jz .L1
        jmp .L2
.L1:
        .... code for S1
        jmp .L3
.L2:
        .... code for S2
        jmp .L3
.L3:
```

Translation Steps: Code Optimization



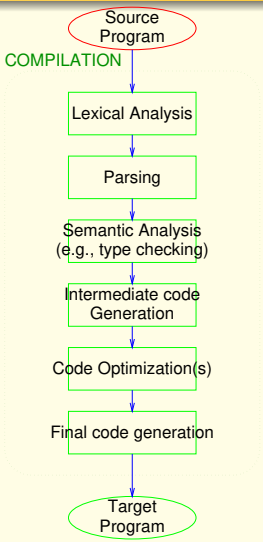
- A key part of modern compilers.
- Transform intermediate code while preserving program behavior.
- Improve execution time, code size, or other resource usage.

Translation Steps: Final Code Generation



- Map instructions in the intermediate code to specific machine instructions.
- Supports standard object file formats.
- Generates sufficient information to enable symbolic debugging.

Translation Steps: Final Code Generation Example




```
    R1 ← mem(m)
    cmp R1, 0
    jz .L1
    jmp .L2
.L1:
    .... code
for S1
    jmp .L3
.L2:
    .... code
for S2
    jmp .L3
.L3:
```

$\Longrightarrow$

```
    movl 8(%ebp), %esi
    testl %esi, %esi
    jne .L2
.L1:
    .... code for S1
    jmp .L3
.L2:
    .... code for S2
.L3:
```

Interpreters and Compilers

- *Compiler:* translates a program into executable code before it runs
  - C++ is commonly compiled to native machine code
- *Interpreter:* executes a program from its source, AST, or another intermediate representation
  - Sash will parse and type-check programs, then interpret them
- *Hybrid:* compiles to bytecode for interpretation or JIT compilation
  - Java uses portable JVM bytecode
- A property of an implementation, not necessarily of the language

Broader Applications of Languages

- Command Interpreters: `bash`, ksh, PowerShell, ...
- Programming: Java, Python, C++, Rust, Go, Haskell, Scala, OCaml, ...
- Document Structuring: LaTeX, HTML, RTF, `troff`, ...
- Page Definition: PDF, PostScript, ...
- Databases: SQL, ...
- Hardware Design: VHDL, Verilog, ...
- Domain-Specific Languages (DSL)

Course Topics

- *Lexical analysis:* Regular expressions, translation to finite-state automata (deterministic or non-deterministic), and regular expression tools.
- *Parsing:* Context-free grammars, push-down automata, parser generation from CFGs, and abstract syntax trees (ASTs).
- *Semantic analysis:* Type systems, unification and type inference, type checking.
- *Runtime techniques:* Memory allocation, memory management, garbage collection techniques, exceptions, and closures.
- *Static analysis and optimization:* Data flow analysis, abstract interpretation, and optimizations based on them.
- *Dynamic analysis and instrumentation:* Applications to debugging, runtime error detection, automated testing and vulnerability analysis.

Course Organization and Logistics

Sash: A Sane Shell

- Existing shells: convenient for commands, but
  - Confusing and sometimes bizarre syntax
  - Awkward expressions and arithmetic
  - Awkward conditions and control-flow constructs
- Sash
  - The convenience of a shell for commands
  - Clean programming-language syntax, types and functions
  - A sufficiently rich, usable implementation

Why a Shell?

- One project spanning almost every part of the course
  - Lexical analysis and parsing
  - Type inference and type checking
  - Evaluation and runtime organization
  - Processes, pipelines and I/O
- Immediate feedback from using the language during development
- Interacting design choices
  - Enough room to make the project your own

Hash: A Hybrid Shell

- A typed shell developed in preparation for this course
- Binary available from the course web page
  - Exploring possibilities; answering general semantic questions
- *Hash and Sash are different languages*
  - Hash: not the specification for Sash
  - Reproducing Hash behavior: likely an incorrect project

Agentic Software Engineering

Software development in which AI agents carry out multi-step engineering tasks under human direction and review

- Install and run a coding agent on your laptop
  - E.g., Codex or Claude Code
  - A paid subscription to one agent service will be required
- Preferred setup: inside a Linux VM or another effective sandbox
  - Isolation from personal files and the host system
- Ask the agent for help installing itself and setting up the VM or sandbox

How to Use AI Tools

- AI as tutor and design partner
  - Learn unfamiliar concepts and tools
  - Explore and refine your design
  - Evaluate implementation choices
  - Discuss defects and alternatives in depth
- Agents for low-level work; your attention on higher-level thinking
- Not merely a subordinate asked to complete your work

You Remain Responsible

- *You must understand the code 100% and be able to defend it*
- Otherwise
  - No control over whether it passes our tests
  - Difficulty with exam questions about the project
  - Inability to explain your own code
- Possible detailed oral review of your code

Course Logistics

The course web page is the authoritative source for all course logistics and current course information.

- Text: the "Dragon Book," any edition
- Approximate grading weights
  - Assignments: 35%
  - Exams: 65%
- Consult the course web page regularly for details and updates

AI Tools and Academic Honesty

- AI tools: allowed and expected
- Submitted design and implementation: your responsibility
  - Direct the work
  - Understand the resulting code 100%
  - Be prepared to explain and defend every part of it
- Inability to explain your code is grounds for initiating an academic dishonesty case
- Prohibited copying from
  - Another person
  - The Internet
  - Outside implementations
  - Previous offerings
- *Penalty for academic dishonsety: F grade*